



THE UNIVERSITY *of* EDINBURGH

Edinburgh Research Explorer

Optimization of a parallel permutation testing function for the SPRINT R package

Citation for published version:

Petrou, S, Sloan, T, Mewissen, M, Forster, T, Piotrowski, M & Dobrzelecki, B 2010, 'Optimization of a parallel permutation testing function for the SPRINT R package', Paper presented at 19th ACM International Symposium on High Performance Distributed Computing, Chicago, United States, 20/06/10 - 25/06/10 pp. 516-521. <https://doi.org/10.1145/1851476.1851551>

Digital Object Identifier (DOI):

[10.1145/1851476.1851551](https://doi.org/10.1145/1851476.1851551)

Link:

[Link to publication record in Edinburgh Research Explorer](#)

Document Version:

Publisher's PDF, also known as Version of record

Publisher Rights Statement:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

General rights

Copyright for the publications made accessible via the Edinburgh Research Explorer is retained by the author(s) and / or other copyright owners and it is a condition of accessing these publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy

The University of Edinburgh has made every reasonable effort to ensure that Edinburgh Research Explorer content complies with UK legislation. If you believe that the public display of this file breaches copyright please contact openaccess@ed.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.



Optimization of a parallel permutation testing function for the SPRINT R package

Savvas Petrou
Edinburgh Parallel
Computing Centre
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh, EH14 1HZ, UK
spetrou
@epcc.ed.ac.uk

Thorsten Forster
Division of Pathway Medicine
University of Edinburgh
Medical School
The Chancellor's building
49 Little France Crescent
Edinburgh, EH16 4SB, UK
thorsten.forster
@ed.ac.uk

Terence M Sloan
Edinburgh Parallel
Computing Centre
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh, EH14 1HZ, UK
t.sloan
@epcc.ed.ac.uk

Michal Piotrowski
Edinburgh Parallel
Computing Centre
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh, EH14 1HZ, UK
mpiotrow
@staffmail.ed.ac.uk

Muriel Mewissen
Division of Pathway Medicine
University of Edinburgh
Medical School
The Chancellor's building
49 Little France Crescent
Edinburgh, EH16 4SB, UK
Muriel.Mewissen
@ed.ac.uk

Bartosz Dobrzelecki
Edinburgh Parallel
Computing Centre
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh, EH14 1HZ, UK
bartosz
@epcc.ed.ac.uk

ABSTRACT

The statistical language R and Bioconductor package are favoured by many biostatisticians for processing microarray data. The amount of data produced by these analyses has reached the limits of many common bioinformatics computing infrastructures. High Performance Computing (HPC) systems offer a solution to this issue. The Simple Parallel R INterface (SPRINT) is a package that provides biostatisticians with easy access to HPC systems and allows the addition of parallelized functions to R. This paper will present how we added a parallelized permutation testing function in R using SPRINT and how this function performs on a supercomputer for executions of up to 512 processes.

Categories and Subject Descriptors

D.1.3 [Software]: Programming techniques—*Parallel programming*; G.4 [Mathematics and Computing]: Mathematical Software—*Parallel and vector implementations*

General Terms

Algorithms, Performance, Experimentation

Keywords

HPC, MPI, Permutation, Microarray

1. INTRODUCTION

Analyses of post genomic data are requiring increasingly large amounts of computational processing power and memory to complete. A popular free statistical software package for carrying out this data analysis is R [1]. At the University of Edinburgh, EPCC along with the Division of Pathway Medicine (DPM), designed and built a prototype package for R, called SPRINT [2], which parallelized a key statistical correlation function of important use to genomic analysis. This prototype successfully demonstrated that parallelized statistical functions could be interfaced with R, providing biologists with an easy route into HPC.

The aim of SPRINT is to require minimal HPC knowledge, minimal changes to existing R scripts, and yet give maximum performance. The package provides an *interface* to HPC and a *library* of parallel functions. This paper focuses on the work undertaken to extend the library through the addition of a parallel permutation testing function.

The SPRINT project [3] carried out a user requirements survey [4] to collect information from the bioinformatics community on which R functions are causing bottlenecks when processing genomic data as well as those that are currently intractable on standard desktop machines. Some of these functions are suited to large-scale distributed-memory machines and this paper is focused on implementing one of these functions. In the SPRINT user requirements survey, respondents were asked to list the five R functions they would consider most useful for inclusion in a parallel R function library. The *mt.maxT* function [5] from the *multtest* package [6] was the 4th most requested function.

1.1 Other parallel implementations

The R package, as it is currently being released, has no built-in parallel features. A few open source groups from the R community developed various packages [7, 8, 9, 10, 11] in an effort to enable R to run in parallel. These packages can execute simple parallel functions with no data dependencies. The primary objective of SPRINT is to contribute to this effort by adding functionality beyond the limits of the available parallel packages.

The existing packages offer tools for applying various forms of parallelism in R codes. However, many R functions, like *mt.maxT*, are implemented in the C language with only their interface implemented in R. The most commonly used approach in these cases is to perform multiple executions on subsets of the dataset, the iteration count, or both. By doing so the memory demand and the run time of each individual execution are reduced to a reasonable level. Although this workaround can be applied in many cases, the partial data produced have to be reduced and processed in order to be transformed into the expected output.

In SPRINT the functions are implemented at C level and execute in parallel with minimal changes to the R code. The results are reduced and transformed to the expected output before they are returned to the user, thus requiring no further processing. In addition, processes can communicate and exchange data, which enables data to have dependencies.

2. THE PERMUTATION FUNCTION

2.1 The serial version: $mt.maxT$

In statistical analysis, a permutation test is one of the methods used for computing the statistical significance of data. This value is a measure of how likely it is to obtain data with the same properties as the one being tested, should the experiment be performed again. This value is known as the p -value. The R *mt.maxT* function computes the adjusted p -values for step-down multiple testing procedures [5], as they are described in Westfall and Young [12].

The function supports two types of permutation generators. A *random* permutations generator and a *complete* permutations generator. Moreover, the function supports six different methods for statistics, used for testing the null hypothesis of no-association between the class labels and the variables. For all the methods both generators are implemented. Furthermore, for all combinations of method/generator the user can either choose to save the permutations in memory before the computations take place or compute them on the fly. Considering all these options, there are 24 possible combinations of generator/method/store. Four of the statistics methods are similar in nature and use the same implementation of generators. Figure 1 shows how many combinations are implemented in R.

The six supported statistic methods are the following:

- **t**: Tests based on a two-sample Welch t-statistics (unequal variances).
- **t.equalvar**: Tests based on two-sample t-statistics with equal variance for the two samples.

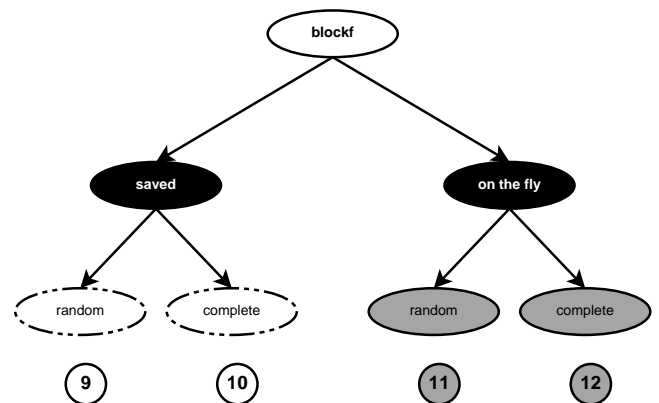
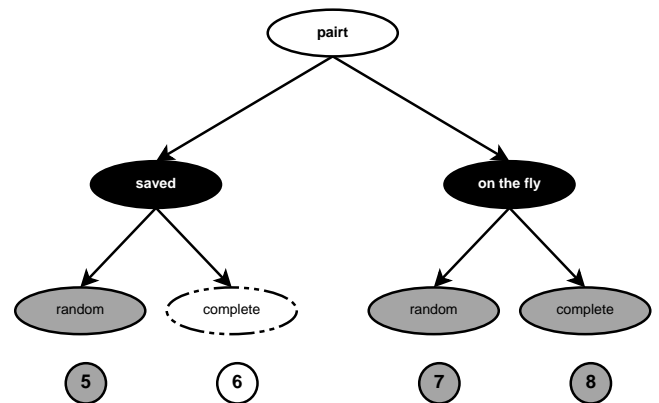
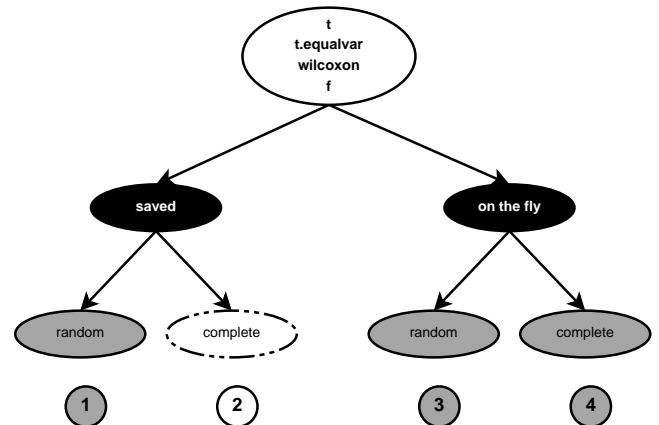


Figure 1: The `mt.maxT` generator/method/store combinations.

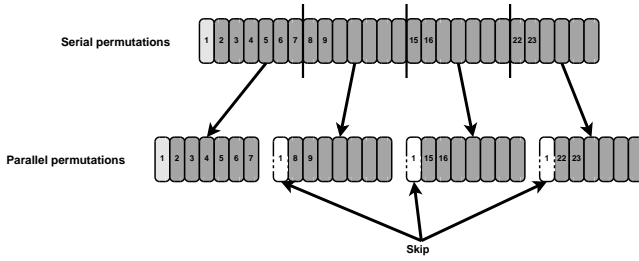


Figure 2: How permutations are distributed among the available processes.

- **Wilcoxon:** Tests based on standardized rank sum Wilcoxon statistics.
- **f:** Tests based on f-statistics.
- **Pair-t:** Tests based on paired t-statistics.
- **Block-f:** Tests based on f-statistics which adjust for block differences.

For *complete* permutations the code never stores the permutations in memory (combinations 2, 6, 10 in Figure 1). Although the option is available, it is implemented using the on-the-fly generator of permutations. For the *Block-f* statistics method the permutations are never stored in memory due to the huge amount of permutations produced. The option is available but the code is again implemented using the on-the-fly generator. The distinct combinations the code is implementing are therefore eight (combinations 1, 3, 4, 5, 7, 8, 11 and 12, in Figure 1).

2.2 The parallel version: *pmaxT*

The parallelism is introduced by dividing the permutation count into equal chunks and assigning them to the available processes. At the end of the computational kernel each process will have gathered a part of the observations needed to compute the raw and adjusted *p*-values. These observations are gathered on the master process where the *p*-values are computed and returned to R.

To be able to reproduce the same results as the serial version, the permutations performed by each process need to be selected with caution. Figure 2 shows how the permutations are distributed among the available processes. The first permutation depends on the initial labelling of the columns and it is thus *special*. This permutation only needs to be taken into account once by the master process. The remaining processes skip the first generator permutation (from both the complete and random generators). Moreover, the generators need to be forwarded to the appropriate permutation. To deal with this the interface of the generators was altered and an additional variable is passed to the initialization function. Depending on the value of this variable, the generators *waste* a number of cycles and forward to the appropriate permutation.

The parallel implementation executes the following steps:

- **Step 1:** The master process executes a pre-processing step to check the input parameters and also transform a few to another format.
- **Step 2:** All processes, apart from the master process, allocate memory to accept the input parameters. The master process needs to broadcast the lengths of the string parameters first. All the scalar integer options are also broadcast for convenience. These values are received in a statically allocated buffer vector (this is necessary to ensure they can be received). All dynamically allocated memory needed is then allocated, initialised and checked.
- **Step 3:** A global sum is performed to synchronize all processes and ensure that the necessary memory is allocated. This memory includes the input parameters, the input data and also the memory to store the final results.
- **Step 4:** Each process computes how many permutations it needs to execute and also initializes its generator to the appropriate permutation. Then it computes the local observations.
- **Step 5:** The master process gathers the partial observations and computes the raw and adjusted *p*-values. The computed values are saved in a memory space allocated by the pre-processing script at the R level. This is necessary in order for the values to be returned back to R when the computations are finished.
- **Step 6:** All processes free their dynamically allocated memory.

The interface of the *pmaxT* is identical to the interface of *mt.maxT*. All functionality was successfully ported to the parallel version:

```
pmaxT(X, classlabel, test = "t",
      side = "abs", fixed.seed.sampling = "y",
      B = 10000, na = .mt.naNUM, nonpara = "n")
```

Compared to:

```
mt.maxT(X, classlabel, test = "t",
        side = "abs", fixed.seed.sampling = "y",
        B = 10000, na = .mt.naNUM, nonpara = "n")
```

Parameters `test`, `side`, `fixed.seed.sampling`, `B`, `na` and `nonpara` are optional. If omitted, the default values shown above are used. The description of the input parameters follows.

- **X** : The input dataset array.
- **classlabel** : The class labels of the columns of the input dataset.

Table 1: Profile of *pmaxT* implementation.

Process count	Pre-processing	Broadcast parameters	Create data	Computational kernel	<i>p</i> -values computations	Speedup	Speedup (computations)
1	0.260	0.001	0.010	795.600	0.002	1.00	1.00
2	0.261	0.004	0.012	406.204	0.884	1.95	1.95
4	0.259	0.009	0.013	207.776	0.005	3.82	3.82
8	0.260	0.013	0.013	104.169	0.489	7.58	7.63
16	0.259	0.015	0.013	51.931	0.713	15.03	15.32
32	0.259	0.017	0.013	25.993	0.784	29.40	30.60
64	0.259	0.020	0.013	13.028	0.611	57.11	61.06
128	0.259	0.023	0.013	6.516	0.662	106.48	122.09
256	0.260	0.024	0.013	3.257	0.611	190.99	244.27
512	0.260	0.028	0.013	1.633	0.606	313.09	487.20

- **test** : The method for statistics, used for testing the null hypothesis.
- **side** : The type of rejection region. Available options are **abs** for absolute difference, **upper** for the maximum and **lower** for the minimum.
- **fixed.seed.sampling** : The choice between computing the permutations on the fly or save all permutations in memory prior to computations. Available options are **y** (yes) for the on the fly generator and **n** (no) for storing them in memory.
- **B** : The number of permutations. When this value is set to 0, the code will try to perform the complete permutations of the data. In case the complete permutations exceed the maximum allowed limit the user is asked to explicitly request a smaller number of permutations.
- **na** : The code for missing values. All missing values will be excluded from the computations.
- **nonpara** : The option for non-parametric test statistics. Available options are **y** for yes and **n** for no.

3. BENCHMARK RESULTS

3.1 HECToR

The parallel functions were tested and benchmarked on the UK National Supercomputing service, the HECToR Cray XT system [13]. At that point the system comprised 1416 compute blades, each with four quad core processor sockets. The CPUs are AMD 2.3 GHz Opteron chips with 8 GB of memory. This gives a total of 22,656 active cores with a total of 45.3 TB of memory and a theoretical peak performance of 208 TFLOPS.

3.2 Benchmarks for *pmaxT*

The benchmarks were executed for process counts 1 to 512. The values reported are the minimum measured timings obtained from 5 executions.

The performance and scaling benchmarks for *pmaxT* execute a permutation count of 150,000 on a dataset of 6,102 rows (genes) and 76 columns (samples). Table 1 shows the time spent on each of the five main sections of the code. Figure 3 shows a graphical representation of the scaling. Figure 4 shows how the speedup of the current parallel version compares to the optimal speedup.

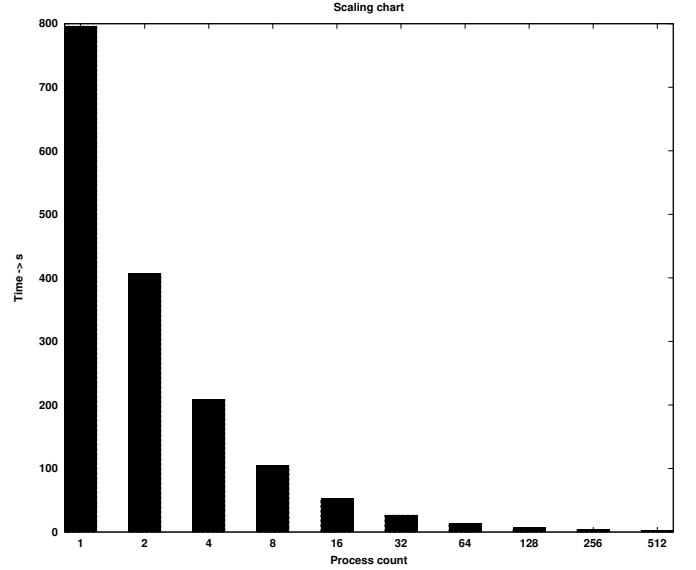


Figure 3: Elapse time of *pmaxT*.

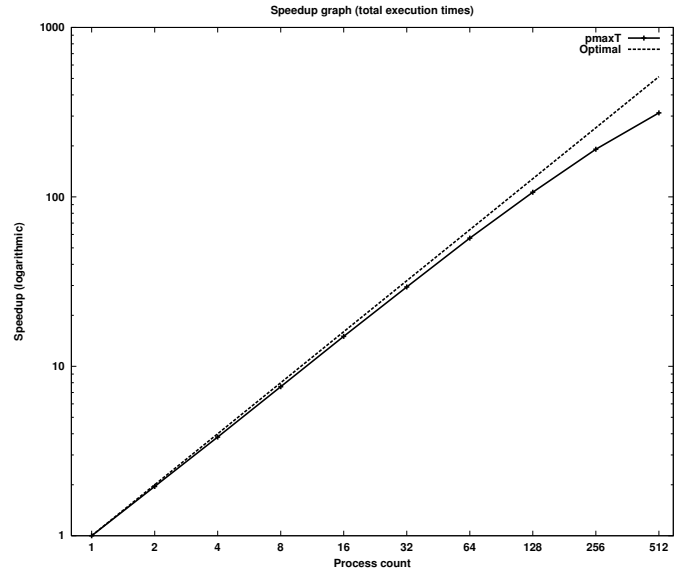


Figure 4: Speedup of *pmaxT* (in logarithmic scale).

Table 2: Processing time of *pmaxT* for various input datasets and permutation counts.

Input array dimension and size (genes \times samples)	Permutation count	Total runtime (in seconds)	Serial runtime (approximation) (in seconds)
36,612 \times 76 21.22MB	500,000	73.18	20,750 (6 hours)
36,612 \times 76 21.22MB	1,000,000	146.64	41,500 (12 hours)
36,612 \times 76 21.22MB	2,000,000	290.22	83,000 (23 hours)
73,224 \times 76 42.45MB	500,000	148.46	35,000 (10 hours)
73,224 \times 76 42.45MB	1,000,000	294.61	70,000 (20 hours)
73,224 \times 76 42.45MB	2,000,000	591.48	140,000 (39 hours)

In addition, benchmarks were executed to measure the amount of time the new parallel implementation needs to execute a very high permutation count. Table 2 shows the run time of different input datasets and permutations counts. All executions were performed with 256 processes.

The run times for the serial implementation in Table 2 are not actual measured timings but an approximation of the real run time. Executions with lower permutation counts (1000, 2000 and 3000 permutations) showed a linear increase in run time as the permutation count increases. According to these results the approximated run times were calculated.

4. DISCUSSION

The results from the benchmarks show a very good scaling behavior. As the number of processes increases, the amount of time needed by the function reduces linearly. However, for very large process counts, the overhead from broadcasting and transforming the input dataset may consume a significant percentage of the total run time and affect the overall scaling (see last two columns of Table 1).

Examining the scaling of the computational kernel alone, it can be seen how well the implementation performs as the process count increases. When the work load of the computations is sufficiently high, the amount of time spent to broadcast and transform the data will be small enough to not affect the overall scaling.

The main barrier for the permutation testing function is the permutation count. As the count of requested permutations increases, the run time becomes excessively costly. The parallel implementation distributes the permutations among the available processes and makes it possible to perform high permutation counts in reasonable run times. When the permutations are generated on the fly, the implementation demands no extra memory in order to perform a higher permutation count. As long as the input dataset can be stored in memory, the implementation can execute a count that is limited only by the precision of the underlying CPU architecture.

Furthermore, the faster execution times of the parallel implementation help with reducing the risk of failures. Long executions have higher risks due to system failures, thus an implementation that performs the same amount of work faster is beneficial.

5. CONCLUSIONS

This paper presents how an R permutation testing function was parallelized for inclusion to the SPRINT package. The scaling of the parallel implementation was tested and the results showed a close to optimal scaling. In addition, the function is able to perform a much higher permutation count within a reasonable run limit.

The success of the parallelization process verifies that SPRINT can be effectively used in order to add parallelized functions to the R statistical language with minimal impact on the user interface.

5.1 Future work

Taking into consideration the information gathered by the user requirements survey [4], more functions can be parallelized and added to the SPRINT package. Along with the new functionality the *interface* of SPRINT can be extended in order to enable developers and scientists to add their own parallel functions with less effort.

Although the parallel implementation of the permutation testing function performs well, a few additional changes can be made. These changes are:

1. The current implementation performs an array transposition on the input dataset. For this transformation a new array is allocated. Algorithms for in-place non-square array transposition exist that are able to perform this step without the need of additional memory.
2. The *string* input parameters can be replaced with scalar integer values before are broadcast to all processes. Scalar parameters are easier and faster to broadcast and handle.

6. ACKNOWLEDGMENTS

Funding : Wellcome Trust, Grant WT086696MA and the NAG dCSE Support service.

7. ADDITIONAL AUTHORS

Additional authors: Peter Ghazal (Division Of Pathway Medicine, University of Edinburgh Medical School, email: p.ghazal@ed.ac.uk), Arthur Trew (EPCC, School Of Physics & Astronomy, University of Edinburgh, email: A.Trew@epcc.ed.ac.uk) and Jon Hill (Applied Modelling and Computation Group, Department of Earth Science and Engineering, Imperial College London, email: jon.hill@imperial.ac.uk).

8. REFERENCES

- [1] The R package, "The R Project for Statistical Computing." Available at : <http://www.r-project.org/>, accessed 20 February 2010.
- [2] Jon Hill, Matthew Hambley, Thorsten Forster, Muriel Mewissen, Terence M Sloan, Florian Scharinger, Arthur Trew and Peter Ghazal, "SPRINT: A new parallel framework for R," *BMC Bioinformatics*, December 2008.
- [3] SPRINT's website, "SPRINT: A new parallel framework for R." Available at : <http://www.r-sprint.org/>, accessed 20 February 2010.
- [4] M. Mewissen, "SPRINT - User Requirements Survey Results." Technical report from Wellcome Trust, Grant WT086696MA, July 24th 2009.
- [5] Y. Ge, S. Dudoit and T. P. Speed, "Resampling-based multiple testing for microarray data hypothesis," vol. 12, pp. 1–77, June 2003.
- [6] The multtest R package, "multtest: Resampling-based multiple hypothesis testing." Available at : <http://cran.r-project.org/web/packages/multtest/index.html>, accessed 20 February 2010.
- [7] NWS, "NetWorkSpaces for R." Available at : <http://nws-r.sourceforge.net/>, accessed 3 March 2010.
- [8] Rmpi, "Rmpi for R." Available at : <http://www.stats.uwo.ca/faculty/yl/Rmpi/>, accessed 3 March 2010.
- [9] R/parallel, "R/parallel framework." Available at : <http://www.rparallel.org/>, accessed 3 March 2010.
- [10] papply, "papply - R parallel apply function." Available at : <http://math.acadiau.ca/ACMMaC/software/papply.html>, accessed 3 March 2010.
- [11] SNOW, "Simple Network of Workstations for R." Available at : <http://www.stat.uiowa.edu/~luke/R/cluster/cluster.html>, accessed 3 March 2010.
- [12] Westfall, P.H. and Young, S.S., "Resampling-Based Multiple Testing: Examples and Methods for P-Value Adjustment.," p. 340, 1993. Wiley, New York.
- [13] HECToR's user support, "HECToR : Hardware overview." Available at : <http://www.hector.ac.uk/service/hardware/>, accessed 20 February 2010.